Artificial Neural Networks and Deep Architectures, DD2437

Short report on lab assignment 2

Radial basis functions, competitive learning and self-organisation

Rahul Shah, Nicolas Essipova, and Love Marcus

February 8, 2019

1 Main objectives and scope of the assignment

Our major goals in the assignment were

- to design RBF and SOM networks,
- to try out the concepts of competitive learning and self-organization,
- to try out unsupervised learning.

Our philosophy has been to build everything that feels relevant ourselves. We have for example built the functions for RBF but relied on numpy and scipy for vector calculus, statistics and other fundamental operations.

2 Methods

We have used MATLab for parts of the assignment and python for the rest.

3 Results and discussion - Part I: RBF networks and Competitive Learning

3.1 Batch mode training using least squares

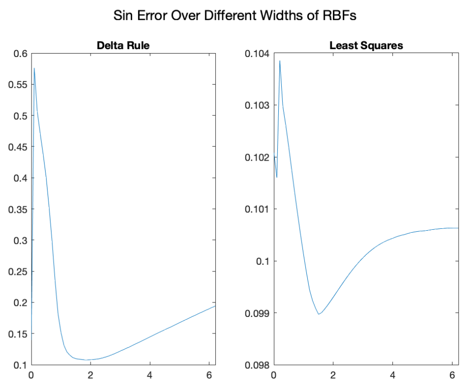
For the least squares solution, we have implemented the least squares algorithm using the built in MATLab solver to find the optimal weights. When the number of hidden nodes was above 15, we got error messages saying that the rank of the matrix was too high to perform least squares and that a significant amount of error would occur in the calculations of weights.

To find the number of nodes where the residual absolute error was below a certain bound, we varied the number of hidden nodes and averaged the absolute residual error over 1000 iterations. Once we did this, we saw that the error dramatically decreases as more hidden nodes are added. The average absolute residual error for the sin function drops below .1 when the 6th node is added, reaches .01 after the 8th node, and below .001 with the 10th node. However, for the square function, the error never drops below .2639.

To reduce the residual error to 0 we can treat the problem as one of classification and not regression as the function only varies between two values. As the square function is periodic we can place the nodes with the same periodicity as the function.

3.2 Function approximation with RBF networks

When first testing out the RBF networks, we want to know the optimal amount of hidden nodes which produce the least amount of error. We chose to stick with the average absolute residual error as our error estimate for evaluating all RBF models. After running 1000 iterations for each hidden nodes ranging from 1 to 75, we deduced that 15 hidden nodes was the best for the delta rule while 13 nodes was the best for the least squares method. After evaluating this, with these optimal number of nodes, we ran different widths on the interval $[0, 2\pi]$ increasing by .1. We found that the delta rule works best with a width of 1.8 and the least squares method prefers a variance of 1.6. This shows that the RBF units prefer to have some overlap when placing the nodes. However, too much overlap and the RBF approximation error starts to increase, as can be seen in Figure 1.



Figur 1: Average absolute residual error over different widths of RBF nodes (note that the scale is different on the y-axes).

	Clean	Noisy
Least Squares	.0017	.099
Delta Rule	.036	.108

Tabell 1: Performance on Noisy & Clean Datasets

In terms of placing RBF nodes in the input space, we chose to place them uniformly depending on the number of hidden nodes were specified. This seemed to work quite well as we compared randomly placing the nodes against placing them uniformly. Like above, we averaged the absolute residual error over 1000 iterations, each time choosing random centers for the RBF nodes. We also used the best parameters determined (width, eta, etc.) and kept them the same for both tests. For the delta rule, there was a significant difference as random placement created .2363 against an average of error .109 for uniformly placed nodes. For the least squares rule, there was not as much of a difference between the randomly placed and uniformly distributed nodes, but the uniformly distributed one performed better on average (.101 and .099 respectively).

Comparing the delta rule and least square solution with the optimal parameters on the noisy and clean dataset provided interesting results. Obviously, the clean data sets provided less error than the noisy sets; however, the least squares solution proved to be better than the delta rule in both cases. Check out Tabel 1 for the exact numbers.

Finally, we compared the least squares algorithm against the perceptron trained with backprop. We found that the least squares algorithm performed better on the sin approximation (.00174 vs .0158), but the perceptron approximated the square function better (.0375 vs .274).

3.3 Competitive learning for RBF unit initialization

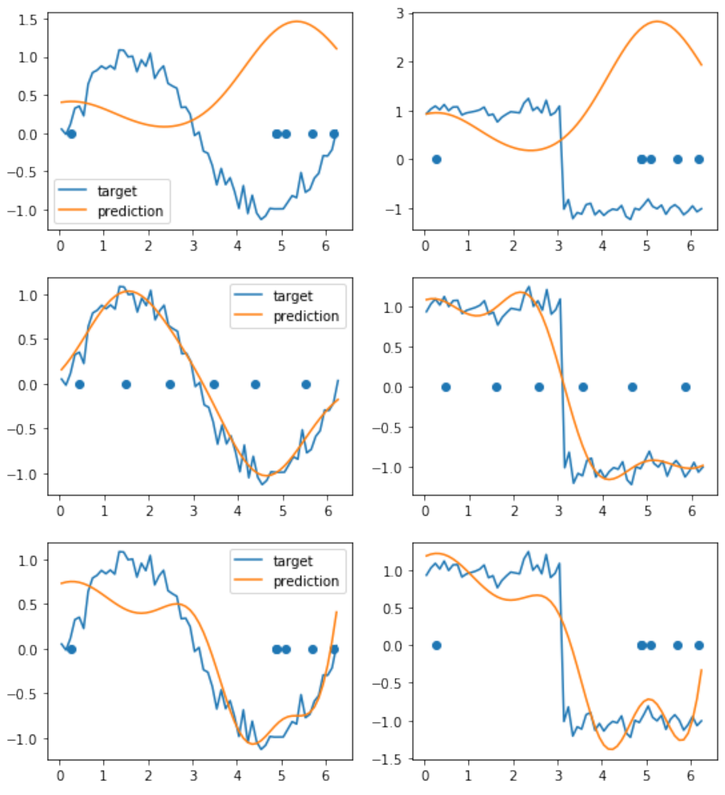
3.3.1 Sinusoidal and square data

We implemented competitive learning in Python that updates the winning node as well as a specified number of nearby nodes. An example of how the CL algorithm adapts the nodes can be seen in Figure 2 where the first two images display the targets and randomly placed nodes and following display the patterns and predictions by an RBF network that was trained after the nodes had been placed by the CL algorithm.

The last two images in Figure 2 display the predictions of an RBF NN that was trained with the randomly placed nodes. The RBF NN gets remarkably close to the actual pattern. In this example, the MSE is 6.9 times higher for the sinus and 2.1 times higher for the square function.

We found that the addition of competitive learning had the most effect for low and semi-high number of nodes, as the CL algorithm allowed the RBF to adapt to the shape of the data. This can potentially lead to over-fitting. We found that increasing the number of winners reduced the tendency to over-fit.

It seems that CL has a higher impact when the data is not evenly spaced as was the case with the sinusoidal and square data sets that we worked with.



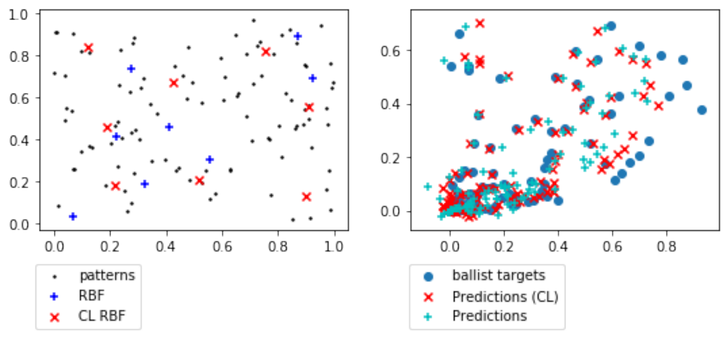
Figur 2: Example of RBF node placement by CL preprocessing

3.3.2 Ballist data

The left image in Figure 3 displays the Ballist test set and 8 randomly placed nodes as well as how they were placed by the competitive learning algorithm. The RBF network were then trained with both the randomly placed nodes and the adjusted nodes and the resulting prediction can be seen in the right image in Figure 3.

The prediction fit is generally much better after node placement with competitive learning with the most notable effect when the number of nodes is low. In Figure 3 the MSE is more than twice as large for the RBF with randomly placed nodes.

One important aspect of applying CL to more than just one winning node is that the drag-along of nearby nodes is one way to eliminate redundant nodes that would never fire. One alternative method of doing this is too look for nodes that never get updated and prune them out.



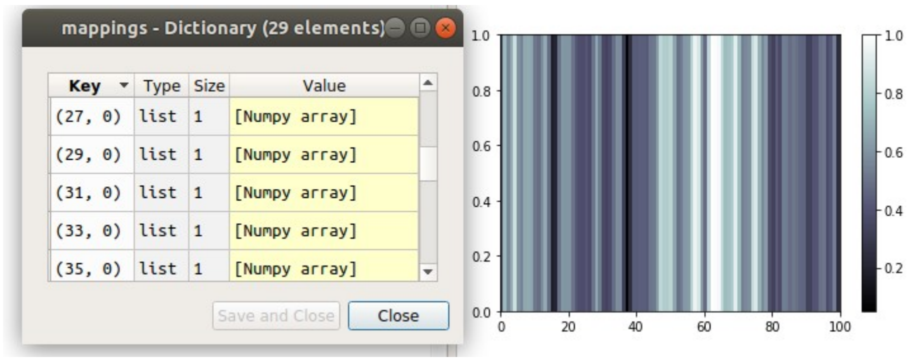
Figur 3: Ballist test set and predictions

4 Results and discussion - Part II: Self-organising maps

4.1 Topological ordering of animal species

The Self-Organizing Map was trained on a 100x1 grid, wherein the animals were organized and placed in adjacency in adherence to their attributes similarities. Thereafter, to view where each animal was positioned, a dictionary was constructed that showed which best matching unit was closest to which animal. In this case, we just had to look at the x-coordinate of said BMU, as y-coordinate was 1 due to the grid being one dimensional. Vast majority of animals had a BMU completely to themselves, but very few shared same BMU.

Examples of the former are dog, giraffe, and pig (BMU positioned at x = 29, 31, and 33 respectively). Examples of the latter where the animals shared the same BMU are beetle, dragonfly, and grasshopper. All three shared the same BMU positioned at x = 97. A list was constructed from the dictionary where each animal was assigned its BMU x-coordinate, then sorted in ascending order. However, it was misleading in certain cases, such as when several animals shared the same BMU as it could be interpreted that one animal was more similar to one animal more than the other depending on its position in the list, which is not true (for example with the beetle, dragonfly, and grasshopper).



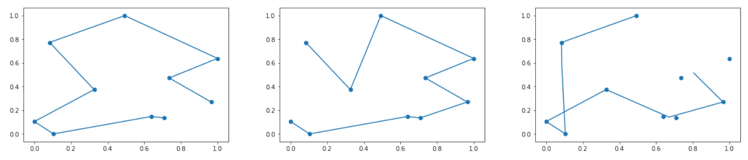
Key	Type	Size	Value
(27, 0)	list	1	[Numpy array]
(29, 0)	list	1	[Numpy array]
(31, 0)	list	1	[Numpy array]
(33, 0)	list	1	[Numpy array]
(35, 0)	list	1	[Numpy array]

Figur 4: The dictionary of the animals (left) and its visualization (right). Each key is the coordinate of the trained grid, containing the animal(s) that are closest to it. In the case of x-coordinate 97, there were three animals (beetle, dragonfly, and grasshopper). The middle three keys (29, 31, 33) contain the array of attributes for dog, giraffe, and pig in order of mention.

4.2 Cyclic tour

We trained a cyclic SOM as instructed and calculated neighbors with modolus when the index passed the start or end of array with the nodes. A sample of the results can be seen in Figure 5. Based on experience of running the script several times (highly unscientific) it seemed to have a higher success rate of finding visibly good solutions when we started off with a higher number of neighboring nodes than suggested.

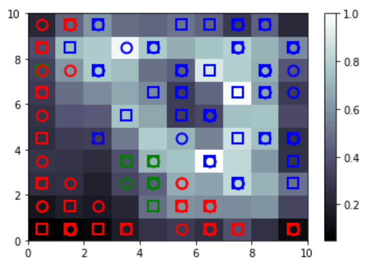
We also tested networks with higher number of nodes and found that they generally worked well. One interesting aspect is the successful pruning of dead nodes.



Figur 5: Examples of the city tour. The image to the left displays a situation where some of the 10 nodes have converged to the same places. The middle image displays a solution with 100 nodes and the right image displays an example of a clearly sub-optimal solution.

4.3 Clustering with SOM

The clustering has been surprisingly accurate as the parties are all grouped in Figure 6. There seem to be mainly one axis of divergence with the green party between the blocks. There is no such apparent difference between the genders.



Figur 6: Clustering of Swedish MP. Parties colored by affiliation (left and right), with green being the environmental party. Squares are males and circles are females.

5 Final remarks

We all found it interesting to build these relatively simple algorithms from scratch. The theoretic background in the assignment document was informative but still short and on point.

We liked the example with the cyclic map as it makes the relationship between the nodes and their weights clear. The data sets with animals and MPs were also interesting examples.

We could maybe have used some better hint on how to format vectors as we had some issues due to diving head first and making some unnecessary mistakes which were sometimes hard to debug.

The first part might have been more interesting if the training set was not uniform over the interval and thus potentially making the CL more interesting.